# Exploitation du graphe de dépendance d'AOSP à des fins de sécurité

Alexis Challande[1,2], Robin David[1] et Guénaël Renault[2,3]
{achallande,rdavid}@quarkslab.com
guenael.renault@ssi.gouv.fr

[1] Quarkslab
[2] LiX, École Polytechnique, Institut Polytechnique de Paris, Inria, CNRS
[3] Anssi

**Résumé.** Contrairement aux *GNU autotools*, le système de *build Soong*, développé par Google, se prête plus favorablement à l'analyse de l'inter-dépendance des cibles de compilations. Utilisées à des fins de sécurité, ces relations de dépendances permettent d'évaluer la propagation d'une vulnérabilités et les composants affectés à travers un graphe, appelé graphe de dépendance unifié. Appliqué à l'*Android Open Source Project*,[4] la construction et l'exploitation de ce graphe permettent de savoir quelles sont les cibles issues d'un fichier. Ces travaux présentent les problématiques techniques liées au calcul de ce graphe et le potentiel offert par son exploitation.

## 1 Introduction

### 1.1 Problème initial

Déterminer dans quelles cibles binaires (bibliothèques ou exécutables) se retrouvent les fichiers sources à la fin de la chaîne de compilation permet d'aider certaines analyses de sécurité comme les études d'impacts ou la recherche de vulnérabilité. Si cette chaîne de compilation est simple, par exemple un fichier source utilisé par un seul exécutable, ce lien est évident. En revanche, les dépendances induites par des bibliothèques statiques compliquent l'étude. Il est bien sûr possible d'établir cette correspondance manuellement, mais cela est fastidieux. Une autre solution est d'effectuer la compilation pour utiliser les informations de débug présentes dans les cibles finales. Cette option nécessite cependant un environnement de *build* valide et d'attendre le temps nécessaire à la compilation. La problématique est donc de trouver une solution automatique permettant d'établir le lien entre des sources et des binaires sans compilation.

4. Agrégat des projets du système Android

### 1.2 Contexte

Le système d'exploitation Android équipe de nombreux appareils (téléphones, voitures...). Il s'articule autour de l'Android Open Source Project (AOSP), composé de plus de 2000 projets disponibles en sources ouvertes. Ces composants vont de projets largement répandus (p. ex. *curl*) à des développements spécifiques pour des appareils particuliers (p. ex. *msm8x09* — un *SoC* Qualcomm) et couvrent l'ensemble du spectre des couches logicielles rencontrées dans un système d'exploitation où de nombreux composants interagissent. La Figure 1 rappelle quelques chiffres clés du projet. Le temps de compilation d'Android 11, la dernière version du système, illustre l'ampleur d'AOSP.

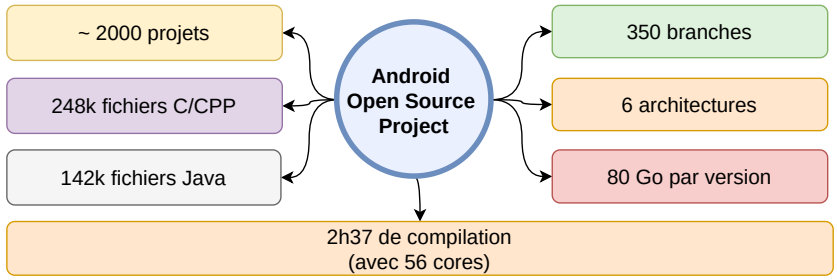


**Fig. 1.** Statistiques sur AOSP

Google publie mensuellement des bulletins de sécurité contenant la liste des vulnérabilités corrigées dans Android. Ces informations, couplées à la possibilité d'utiliser AOSP sur plusieurs architectures et la diversité de ses composants, en font une base d'expérimentation riche pour de la recherche en sécurité.

### 1.3 État de l'art

Les travaux sur les systèmes de *build* dans la littérature se concentrent sur la découverte de dépendances implicites ou redondantes. En 2014, un graphe de dépendance construit à partir de l'analyse de *Makefile* est utilisé à ces fins [5]. Pour retrouver certaines incohérences, ce graphe est ensuite comparé avec un graphe construit à partir des *imports* des fichiers sources [6]. Finalement, en 2020, `VeriBuild` [2] formalise *Unified Dependency Graph* (UDG — graphe de dépendance unifié). Pour augmenter la fiabilité et la complétude du graphe, ces derniers sont complétés par l'instrumentation de l'environnement de compilation.

Certains travaux se focalisent sur Android [1] et plus particulièrement sur les différences entre les versions 6 et 7 qui utilisent encore *GNU autotools*. Cependant, Google a entamé une transition vers *Soong* à partir d'Android 7. Les précédents travaux ne sont désormais plus applicables. Cette étude utilise ce nouveau système pour construire statiquement le graphe, et ce, beaucoup plus rapidement que l'état de l'art.

## 2 *Soong* et *blueprint*

| | *GNU autotools* | *Soong* |
|---|---|---|
| Fichiers | Makefile | blueprint |
| Syntaxe | «Makefile» | «JSON-like» |
| Unité de compilation | règle | module |

**Tableau 1.** Comparaison entre *GNU autotools* et *Soong*

Depuis la version 7 (Nougat), le système de *build* d'Android est *Soong* [4]. Il remplace les anciens *Makefile* (Android.mk). Ce système semble uniquement utilisé dans AOSP pour le moment.

Les *blueprints* sont les fichiers de directives de compilation utilisés par *Soong*. Leur syntaxe est un mélange entre celle du JSON et celle de description des *Protocol buffers* [3]. Les principales différences entre les *GNU autotools* et *Soong* sont illustrées dans le Tableau 1.

Les *blueprints* sont exclusivement déclaratifs et toute la logique de construction est gérée par *Soong*. Le Listing 1 montre un extrait simplifié d'un tel fichier, définissant les directives de compilation de la bibliothèque `liblpdump`.[5]

```
1 cc_library_shared {
2   name: "liblpdump",
3   defaults: ["lp_defaults"],
4   shared_libs: [ "libbase", "liblog", "liblp",],
5   static_libs: ["libjsonpbparse",],
6   srcs: ["lpdump.cc", "dynamic_partitions_device_info.proto",],
7 }
```

**Listing 1.** Définition d'un module dans *Soong*

5. Issu de `https://android.googlesource.com/platform/system/extras/+/refs/tags/android-11.0.0_r31/partition_tools/Android.bp#31`

Deux mécanismes sont à noter dans les *blueprint*. Le premier est la possibilité d'utiliser des *defaults*, qui peut s'expliquer comme la mise à jour d'un dictionnaire : les valeurs par défauts sont utilisées si elles ne sont pas redéfinies par le module qui les utilise. Le second est la possibilité d'utiliser des variables dont le fonctionnement se rapproche des macros du language C.

Il est possible d'utiliser les informations contenues dans chacun des *blueprints* d'AOSP pour établir une suite de dépendances des cibles de compilation. En effet, chaque cible définit précisément l'intégralité de ses dépendances, à la fois dynamiques (ligne 4) et statiques (ligne 5) ainsi que les fichiers sources nécessaires à sa compilation (ligne 6).

## 3 Méthode

Nous utilisons un graphe orienté, nommé BGraph (pour *Build-Graph*), pour représenter les dépendances entre les cibles de compilation [2]. Les nœuds du graphe sont soit les cibles (programmes, bibliothèques), soit les fichiers sources. Les arcs représentent la relation entre deux nœuds ; ils sont orientés dans le sens d'utilisation (du fournisseur vers l'utilisateur) et les dépendances de sources (entre un fichier et une cible) sont séparées de celles entre deux cibles.

Le caractère explicite des *blueprints* permet de faire l'hypothèse que le graphe construit par leur analyse est complet. Cela permet de travailler statiquement, et de s'affranchir autant de l'environnement de compilation que du code source en lui même.

La représentation sous forme de graphe permet d'utiliser des algorithmes standards pour extraire l'information souhaitée. Par exemple, les cibles potentielles d'un fichier sont l'ensemble des nœuds vers lesquels un chemin existe depuis la source. La Figure 2 montre une représentation du Listing 1 sous forme de graphe où les liens d'utilisation sont représentés par des arcs.

Pour générer le graphe de dépendance d'AOSP, il faut analyser l'ensemble des fichiers de *build* de la plateforme, puis ajouter un lien pour chaque indication de dépendance. Pour tenir compte de l'évolution des projets d'AOSP et garder des résultats précis, il faut que le graphe et la version d'AOSP correspondent. Nous avons donc créé un graphe pour chacune des versions d'AOSP.

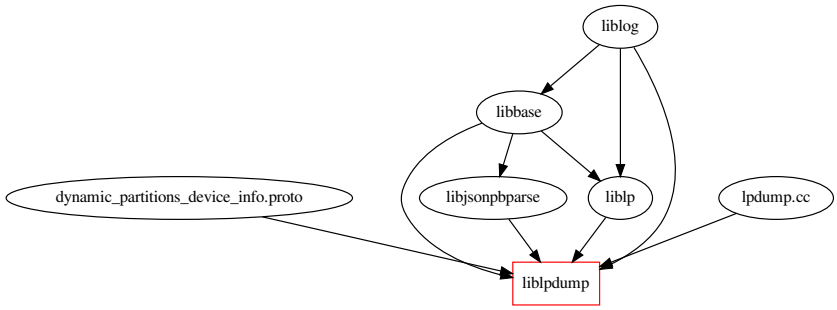


**Fig. 2.** Représentation du Listing 1 sous forme de graphe

## 4 Résultats

Nous avons généré les BGraph pour chaque version d'AOSP à partir de la `7.0.0_r1`, la première à utiliser *Soong*. Cela représente 350 versions jusqu'à la `11.0.0_r31`.

La migration entre l'ancien système de construction *Android.mk* et *Soong* reste incomplète. La dernière version d'AOSP contient encore 1 344 *Android.mk* pour 6 084 *blueprints*. Ces derniers ne sont pas analysés par notre travail et les dépendances définies dans ces fichiers sont ignorées. La précision de BGraph devrait donc s'améliorer avec le temps comme le montre la Figure 3.

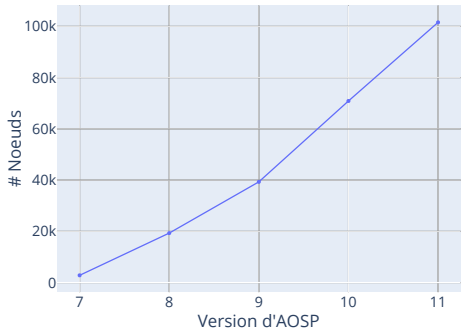


**Fig. 3.** Nombre de nœuds dans les BGraph

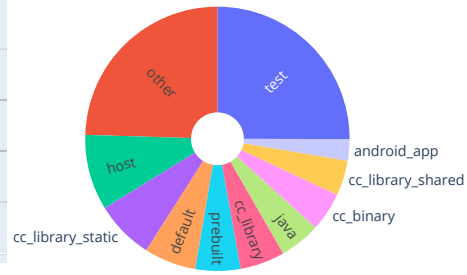


**Fig. 4.** Répartition des nœuds dans un BGraph d'Android 11

Les cibles de compilation d'une version d'Android se répartissent selon le graphe de la Figure 4. Le grand nombre de cibles de tests s'explique car la plupart des modules s'accompagnent d'une logique permettant de tester la validité du code. Environ 10% des cibles d'AOSP concernent du code destiné à l'hôte qui construit et non à l'appareil.

## 5 Cas d'usage

### 5.1 Diffusion de la `CVE-2020-0471`

La vulnérabilité CVE-2020-0471, corrigée dans le bulletin de sécurité de janvier 2021, permet à un attaquant d'injecter des paquets dans une connection Bluetooth et peut conduire à une élévation de privilèges. La vulnérabilité est fixée par le commit `ca6b0a21` [6] s'appliquant à `packet_fragmenter.cc`.

`packet_fragmenter.cc` est utilisé pour la construction de la bibliothèque statique `libbt-hci`. Un système classique de détection de dépendances, utilisant la détection des chargements (*imports*) n'est pas capable d'aller plus loin. Une requête avec BGraph permet de résoudre également les dépendances statiques additionnelles.

```
1  % bgraph query graphs/android-11.0.0_r31.bgraph --src '
       packet_fragmenter.cc'
2            Dependencies for source file
3                packet_fragmenter.cc
4
5    Target         | Type                | Distance
6    ===============|=====================|==========
7    libbt-hci      | cc_library_static   | 1
8    libbluetooth   | cc_library_shared   | 2
9    libbt-stack    | cc_library_static   | 2
10   Bluetooth      | android_app         | 3
```

**Listing 2.** Exemple de requête

Le Listing 2 présente cette requête. Le retour de la commande permet de voir (ligne 8) que seule la bibliothèque bluetooth (`libbluetooth.so`) est un point d'entrée sur le système. On peut aussi voir que l'application Bluetooth d'AOSP (ligne 10) utilise la bibliothèque et par extension le code vulnérable. Dans ce cas, la dépendance semble naturelle, mais l'intérêt est de pouvoir effectuer cette classe de requêtes automatiquement et vérifier l'ensemble des binaires impactés.

6. `https://android.googlesource.com/platform/system/bt/+/ca6b0a211eb39ba85eed60ea740c85d1122fc6bc`

### 5.2 À la recherche des meilleures cibles

Comme montré en Figure 1, AOSP est un projet conséquent. Cette partie montre comment utiliser un BGraph afin de retrouver quels sont les fichiers les plus utilisés dans AOSP, et *in fine*, lesquels seraient les plus importants à analyser.

| Cible | Nbr. Nœuds | Description |
|---|---|---|
| libbase | 3025 | Fonctions classiques pour Android |
| fmtlib | 3027 | Alternative à *stdio* et *iostreams* |
| liblog | 3340 | Bibliothèque de gestion de log |

**Tableau 2.** Dépendances les plus utilisées dans AOSP

En utilisant l'API de BGraph, on peut voir la taille du graphe induit par chacun des fichiers sources d'AOSP. Plus le graphe est important (en nombre de nœuds), plus le fichier est utilisé dans des cibles de compilation différentes. Le tableau 2 liste les trois bibliothèques les plus utilisées dans le système d'exploitation. Un auditeur pourrait prioriser ces bibliothèques pour maximiser son impact. La requête pourrait néanmoins être raffinée, par exemple en ne considérant que les modules ciblant un appareil (et non l'hôte).

## 6 BGraph : un outil d'analyse de graphe

BGraph [7] est l'outil développé afin de générer et d'interroger les graphes de dépendance. Il se présente sous la forme d'un module Python, utilisable en tant qu'utilitaire ou d'API.

*Construction d'un (des) graphe(s).* Pour construire un graphe, BGraph a besoin d'accéder au miroir d'AOSP.[8] L'utilisation d'un miroir local permet d'accélérer les nombreuses requêtes sur les dépôts `git` au prix d'une utilisation d'espace disque accrue. Les projets de la branche choisie sont ensuite partiellement récupérés par un *sparse-checkout* pour ne télécharger que les *blueprints*. Ce *checkout* partiel permet d'économiser de l'espace disque (140 Mo au lieu de 360 Go). Chaque *blueprint* est ensuite analysé syntaxiquement, leur contenu combiné puis transformé en graphe et sauvegardé.

7. `https://github.com/quarkslab/bgraph`
8. Si la méthode est utilisable sur tous les systèmes utilisant *Soong*, l'implémentation dans BGraph est spécifique à AOSP.

*Requêtes dans le graphe.* BGraph accepte les requêtes dans le graphe en utilisant la commande `query`. Des requêtes sont disponibles pour retrouver les dépendances à partir d'une cible ou de fichiers sources et ont une complexité linéaire dans le nombre de nœuds. Les résultats sont présentés au choix au format texte, JSON ou DOT pour visualiser la chaîne de dépendance.

## 7 Limites

Certaines limites de la méthode sont à considérer. La plus importante est qu'elle repose sur l'exhaustivité du système de *build* qui n'est pas garantie. Par exemple, les composants d'AOSP qui utilisent encore *autotools* sont ignorés dans ce travail. De plus, certaines fonctionnalités des *blueprints* ne sont pas prises en compte, comme les dépendances conditionnelles (sources dépendantes de l'architecture cible). Finalement, il n'est pas possible de combiner les approches précédentes [1, 2] avec la nôtre car la syntaxe des *Makefile* n'est pas assez descriptive. À titre d'exemple, il est difficile d'identifier le type de cible construit par une règle dans ces derniers alors que l'information est explicite dans un *blueprint*.

## 8 Conclusion

Ces travaux montrent une résolution du problème d'analyse de la propagation des fichiers sources vers leurs cibles binaires par l'analyse de leur graphe de dépendance. Des exemples d'utilisation de cette approche sont présentés dans la Section 5. D'autres utilisations sont possibles comme l'étude des différences entre deux versions par la comparaison de leurs graphes ou, pour faciliter la rétro-ingénierie de binaires, en éliminant les dépendances connues.

Nous avons appliqué nos travaux sur AOSP car il est possible de l'utiliser pour de nombreuses analyses orientées sécurité. Il est composé de nombreux projets variés ciblant plusieurs architectures sur lesquelles le code source et des informations de sécurité comme les CVE sont disponibles.

## References

1. Bo Zhang, Vasil Tenev, and Martin Becker. Android build dependency analysis. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–4, Austin, TX, USA, May 2016. IEEE.

2. Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang. Escaping dependency hell: finding build dependency errors with the unified dependency graph. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 463–474, Virtual Event USA, July 2020. ACM.
3. Google. Protocol Buffers. `https://developers.google.com/protocol-buffers/`.
4. Google. Soong. `https://android.googlesource.com/platform/build/soong/+/refs/heads/master/README.md`.
5. Xin Xia, David Lo, Xinyu Wang, and Bo Zhou. Build system analysis with link prediction. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1184–1186, Gyeongju Republic of Korea, March 2014. ACM.
6. Bo Zhou, Xin Xia, David Lo, and Xinyu Wang. Build Predictor: More Accurate Missed Dependency Prediction in Build Configuration Files. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 53–58, Vasteras, Sweden, July 2014. IEEE.